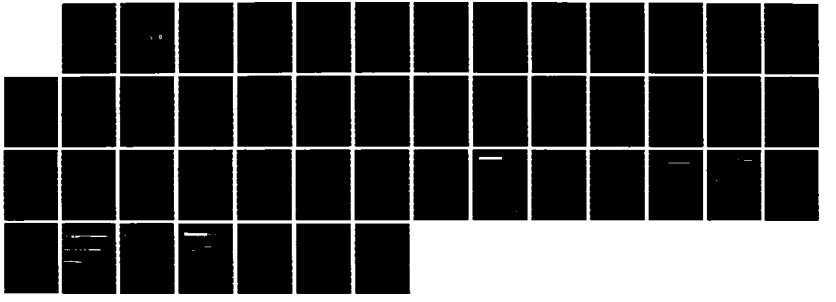
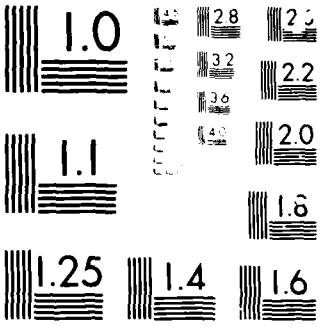
AD-A168 848 COMPUTING ENVIRONMENTS FOR DATA ANALYSIS PART 3 1/1
PROGRAMMING ENVIRONMENTS..(U) STANFORD UNIV CA LAB FOR
COMPUTATIONAL STATISTICS J A MCDONALD ET AL. 21 MAY 86
UNCLASSIFIED LCS-TR-24-PT-3 N00014-81-K-0340 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
2.2
2.0
1.8

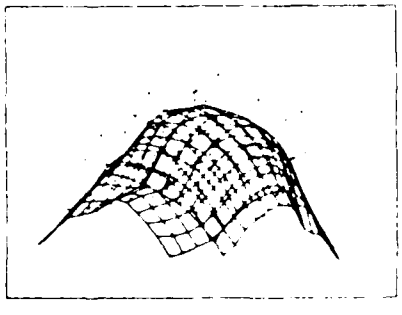
COMPUTING ENVIRONMENTS FOR DATA ANALYSIS: PART 3: PROGRAMMING ENVIRONMENTS

John A. McDonald and Jan Pedersen

Technical Report No. 24

MAY 1986

Laboratory for Computational Statistics



Department of Statistics
Stanford University

Computing Environments for Data Analysis Part 3: Programming Environments *

JOHN ALAN MCDONALD
Dept. of Statistics, University of Washington

JAN PEDERSEN
Dept. of Statistics, Stanford University

May 21, 1986

Abstract

This is the third in a series of papers on aspects of modern computing environments that are relevant to statistical data analysis. In this paper, we discuss programming environments. In particular, we argue that *integrated programming environments* (for example, Lisp and Smalltalk environments) are more appropriate as a base for data analysis than conventional operating systems (for example Unix).

Keywords: *Data Analysis, Workstations, Programming Environments*

*This research was supported by a 1985 Office of Naval Research Young Investigators Award, Dept. of Energy contracts DE-AC03-76F00515 and DE-AT03-81-ER10843 Office of Naval Research contractS N00014-81-K-0340 and N00014-83-K-0472, Office of Naval Research grant N00014-83-G-0121, U.S. Army Research Office contract DAAG29-82-K-0056.

1 Introduction

1.1 Requirements of Data Analysis

How we think about data analysis is strongly influenced by the computing environment in which the analysis is done.

At present, most actual data analysis is done with statistical packages that were designed for the batch-processing environments of 20 years ago. Moreover, nearly all research in new statistical methods tacitly assumes the limitations imposed by batch processing and statistical packages.

The need of data analysis that is most poorly served by statistical packages is *the ability to improvize an analysis on the spot*. Analyzing data often proceeds as a series of solutions to small, easy problems, each of which gives more information about the data and suggests new directions to explore. In the process unexpected things nearly always happen. The practice of data analysis is as much figuring out what question to ask as figuring out how to answer it.

Typical analyses repeat cycles like the following:

- Perform some operation on a data set (i.e. run a program).

  For example, suppose we need a smooth approximation, $\varphi(x)$, to the age distribution of a population. A first choice for $\varphi(x)$ might be something simple, like the fraction of the population within some interval about x.

- Examine the results.

  For this example, we could compare a plot of the curve of $\varphi(x)$ with a histogram of the observed age distribution.

- Choose the next action based on the results.

  After looking at the plot, we may decide to use a $\varphi()$ that is more complex and idiosyncratic, such as one that treats multiples of 5 and 10 differently from other values of x.

We have acquired this view of data analysis as fundamentally improvizational from the work of John Tukey[27,39] and—in the context of computing systems—more immediately from the S system of Becker and Chambers[4,5,6]. From our point of view, the most important contribution of S is the realization that the unpredictable nature of good data analysis demands an *interactive programming language* rather than the fixed set of canned statistical procedures provided by the statistical packages. Many other statistical systems incorporate degrees of programmability, but (as far as we are aware) S goes the farthest in considering itself a programming environment first and only secondarily a collection of statistical primitives.

1.2 Choices for the future

S was developed in the late 70's; it was intended to be used on multi-user minicomputers like PDP 11's and Vaxes running the Unix operating system, with a pen-plotter or graphics terminal for viewing plots. In previous papers, we have argued that, in the mid 80's, a *single-user graphics workstation* is the appropriate hardware for new data analysis systems [25,26]. The topic of this paper is software—what style of programming environment is the best choice as a base for research in future systems like S. The basic alternatives are a *conventional operating system* or an *integrated programming environment*.

Examples of conventional operating systems are Unix, DEC's VMS, DEC's TOPS-20, and IBM's VM/CMS. They are distinguished by an emphasis on support for diverse languages in a multi-user environment. Dozens of companies sell workstations with conventional operating systems. They treat a workstation as simply a minicomputer with a display added—that happens to be used by a single person. The operating systems used on these workstations are inherited with little change from the multi-user computers of the 1970's.

The existing integrated programming environments include: Smalltalk [17,19,18,24], InterLisp-D [33,37,40], and Loops [8,35], all originally developed at Xerox PARC, and available on workstations from Xerox and Tektronix, and several similar CommonLisp-based environments, available on workstations from LMI, Symbolics, and Texas Instruments, that had their

origins in the Lisp machine project at MIT [2]. Altogether, integrated programming environments are found on 10–20 models of workstation from about a half dozen manufacturers.

An explanation why there are so many more workstations with conventional operating systems is that the integrated programming environments are a greater departure from the past. The integrated environments are designed to take advantage of the potential of a *single-user* computer with a high-resolution display to encourage an experimental style of programming[32].

The discussion in this paper will no doubt be influenced by our experience, which is primarily with with Unix and Lisp environments, including several years as systems programmers on machines running various versions of Unix and about two years working with the Xerox InterLisp-D environment (Pedersen) and the Symbolics Lisp Machine environment (McDonald).

When we began working with Lisp machines in 1984, these processors were typically implemented in discrete (bit-slice) components [25] and, largely as a result, were 2-3 times as expensive as equally powerful workstations with more conventional architectures, for which VLSI microprocessor implementations were available (e.g. Motorola 68000). However, by May 1986, the prices of Lisp machines have dropped to be perhaps 25-50% higher than equivalent 68020-based machines. VLSI implementations of Lisp processors should appear within the next year or two, at which point there should be no significant difference in price.

The remaining sections in this paper will assume that our readers are familiar with some time-sharing, multi-user operating system, to the extent of having written and run small programs (hundreds of lines) in a language like C. The arguments we make will be more meaningful to readers who have written and debugged programs that required linking—that used procedures stored in separately compiled files. It will also be helpful if the readers have used an interactive subsystem based on an interpreted language—like S, Minitab, Macsyma, APL, or Lisp.

2 Programming environments

The *programming environment* is usually taken to mean a *set of tools* for writing, storing, managing, compiling, debugging, analyzing, and running programs written in a language like C or Pascal and a *set of parts* (e.g. subroutine libraries) for building new programs. However, we will use the term "programming" here in a broader sense; we consider any act that modifies the state or behavior of the machine to be a form of programming and we will take an equally broad sense of the meaning of "programming environment".

In this context, the most important thing the programming environment does is to give the user a way to think about what is going on in the machine—*a mental model.* At the lowest level, the machine is a structure built of pieces of metal and silicon, carrying varying voltages and currents. To do anything useful, we need to think and act in terms of some higher level abstractions.

A model can often be described in terms of a set of abstract *objects* and *actions* that can be done to objects [15]. In the various models provided by most environments, typical objects are integers, arrays, files, processes, display devices, and so forth. An action might be addition of integers, compiling a file, interrupting a process, drawing a picture on a display.

Each model is usually associated with a *language.* The language is a formal way of expressing ideas about the objects and actions of the model. It provides a set of primitive objects and actions and—if it can be called a "programming" language—ways to combine the primitive entities to define new, more complex objects and actions.

2.1 Data analysis as experimental programming

In order to compare programming environments, we need to consider what data analysis is like, as a computing activity.

The cycle described in the introduction—run a program, examine the results, and run a new or modified program based on the results—is similar

to debugging, with some important differences. The traditional debugging cycle is an iterative way of satisfying a detailed specification for the properties of a program. The programmer compiles and executes an imperfect version, looks at the results, and then modifies the program to bring it more in line with the specification. Programming environments intended primarily for debugging or *production programming* solve the problem of *implementation* [32].

There are important differences between debugging and data analysis. When we sit down to analyze data we are attempting to come up with a solution to a problem that is not well understood when we begin; there is no well-defined specification. We want to learn new things about a data set, which implies that we have vague ideas of what to expect. *"Good data analysis is highly iterative, responding to important facts observed in the analysis itself"* [4].

A better paradigm for data analysis is *experimental programming* [3,11,30,32]. Environments for experimental programming emphasize solving the problem of *design*. In experimental programming, one starts with a vaguely stated problem and works towards a more clearly defined problem as much as one works towards a solution.

In production programming, one wants to change as little of the existing code as possible and make sure that changes are made only after careful consideration of the consequences. This style of programming is often done by large teams (10's to 100's of programmers) and so the environment needs to restrict the ability of any single programmer to make arbitrary changes.

Experimental programming is usually done by very small groups (one or two programmers), who make frequent, fundamental structural changes, exploring novel approaches to the design. Experimental programming environments are designed to amplify the power of the individual programmer and minimize and defer the constraints on what kinds of changes he can make [31].

Data analysis is also usually done in small groups, rather than large teams. Data analysts spend some time playing with minor variations and inspecting the details of a particular model or view of the data, but intermediate results often lead the analysis to branch off in completely new directions.

The basic argument of this paper is: *Interactive data analysis requires experimental programming and is similar to it in spirit. Therefore, an environment designed to support experimental programming is a good base for future data analysis systems.*

2.2 Requirements

An environment that encourages experimental programming must have: [11]

- Fast turn-around for small changes to large programs.

  If we make a small change to an existing procedure or data structure, the time we have to wait while the machine compiles, links, loads, etc., should be as short as possible.

- Natural mental models for the computing environment.

  It should be easy to understand how the existing system works and how to modify them to get the desired effect. This depends on the programming language(s); it will be easier to understand what's going on if the abstractions provided by the programming language model the way we naturally think about the problem.

- Simple, local changes to large programs are usually sufficient and correct.

  We would like to get the desired effect while making as few changes as possible to the existing source code. This makes the change faster and less likely to generate bugs. Whether this is possible depends on the programming language; in many languages, for example, a change to a data structure will require corresponding changes to all procedures that use that data structure.

  The example mentioned in the introduction requires changes to the system that can be reasonably carried out in conventional "high-level" languages like Pascal or Lisp. Other sorts of programming tasks, particularly those involving real-time response, can only be done by

programming at a lower, assembly language or microcode level. The programming environment should support programming at all levels and should allow free mixing of code at all levels; there should be no barriers to communication between code fragments written in any level.

- Tools for modifying program and data structures correctly.

  In other words, the environment should provide an intelligent editor that makes it easy to find the object we want to edit and prevents us from introducing trivial errors.

- Tools for inspecting the environment (what to change).

  We need tools that give us views of the environment, on both a large and small scale. An editor can be thought of as providing detailed views of procedure objects. It is also important to have tools that let us see larger scale structure, such as a plot that shows the directed graph of all the procedure calls in a program.

3 Conventional Operating Systems

Most graphics workstations are provided with a conventional operating system—Unix or something similar. These operating systems were originally developed for multi-user mini-computers like PDP-11's and Vaxes; the versions that run on workstations are inherited with little change.

Many aspects of the structure of conventional operating systems—that make sense in a multi-user time-shared environment—are not necessary or appropriate for a single-user machine. In particular, the harmonious sharing of one machine by several users imposes restrictions on each programmer that limit improvization.

3.1 Processes and Address Spaces

In this section, we describe a mental model of a conventional operating system that might used by a fairly sophisticated programmer, such as the designer and implementor of a large system like S.

To get the machine to do something, in an operating system like Unix, the user spawns a *process* to carries out the desired computation. The life cycle of a simple process can be summarized:

1. A new, empty, virtual address space is created.

   The virtual address space provides a mapping between the abstract entities of the process (procedures and data objects) and locations in the physical memory of the computer. Most operating systems, designed for multi-user machines, maintain the illusion of an isolated address space for each process. This prevents one user's program from interfering with someone else's. In other words, one programmer's singular value decomposition of a matrix called X should not interfere with another's FFT of an array that also happens to be called X.

   Name conflicts and similar problems also arise on a multi-processing single-user machine, but they can be managed by less drastic means—which, as we will see below, removes a lot of overhead from simple program and data manipulations.

2. A first, automated, initialization occurs as procedure definitions are copied from the file system into the address space (by the loader).

   Each process is the execution of a particular load module or program, which is the result of compiling and linking one or more files holding source code. A process's load module cannot be modified while it is running. If a user decides to change a procedure, the current process must be stopped, the procedure's source file edited and re-compiled, the load module re-linked, and a new process started to execute the modified load module.

   In addition, modifications to a single file often require re-compiling many other files. In operating systems like Unix, most programs are written in a compile-time typed language (like C, Pascal, Fortran, etc), which implies that all procedures that refer to a given type (data structure definition) will need to be re-compiled if the type definition is changed.

3. A second initialization step is controlled explicitly by the programmer; initial contents of data objects are read in from the file system by the user's program.

4. Internal computation proceeds by modifying the state of the data objects in the process's address space.

5. Finally, any results of the computation that will be needed in the future are saved by being written out to the file system.

   When a process stops, its address space disappears—and so do the procedures and data objects the address space contains. The only objects in the environment that persist beyond the life of a process are files. So to have any permanent effect on the user's environment, or to communicate with another process, a process must create or modify a file. (Unix offers a short cut for I/O-based interprocess communication called pipes which identifies the output "file" of one process with the input "file" of another.)

This pattern of computation puts a lot of overhead on incremental changes to either data or procedures. Interesting computation gets done only in step 4. Nearly all the work in steps 1, 2, 3, and 5 is redundant, if, for example, one is only changing a single procedure definition or a small part of a large database, which are the most common operations in both program development and interactive data analysis. To make efficient use of the machine, the programmer must arrange the computation so that the time spent in step 4 is large compared to the time spent in the other steps; otherwise the machine will be spending an excessive number of cycles on housekeeping tasks. The usual strategy in Unix is to try to break up programs and the data they operate on into small, independent modules (the *Software Tools* philosophy [22]), which limits at least some of the initialization effort. Unfortunately, this isn't always possible.

An alternative strategy, used by large sub-systems like S and Macsyma, is to make the program complex, interactive, and adaptable enough so that the process can remain running in step 4 indefinitely. This has disadvantages that we will discuss below.

3.2 Multiple languages

In the previous section, we described a possible mental model of a conventional operating system. That picture is not the full story; to make successful use of a conventional operating system, a programmer has to understand and be able to mentally switch among several different, overlapping, conflicting models of his computing environment, which correspond to the different programming languages. The major types of languages are:

- The (compiled) programming languages

  Examples are C, Pascal, and Fortran. The primitive objects are data structures of the language; primitive actions are the built-in functions of the language. New objects can be defined by defining new data structures; new actions can be defined by defining new procedures.

- The (interpreted) shell language

  The basic objects are files (byte streams) and processes. Each process has one input stream and one output stream. Streams are generic objects and are treated uniformly—actual streams might be files, I/O devices, input or output of another process (pipes). Actions are compiling, linking, loading, executing files; spawning and killing processes. New actions can be defined as small programs in the shell language.

Many programming tasks require dealing with other languages (and models) like a device-independent graphics language, a text-editor, a debugger, or a statistical language (like S [4,5,6]). Each language corresponds to a model of the programming environment that is appropriate for solving problems in a particular domain. Many of these languages (e.g. the Unix shell) started as simple sets of commands and, over time, acquired many features of the conventional programming languages.

Heering and Klint [20] point out several problems with such *multilingual* programming environments: The user is confused by the need to understand and use simultaneously several similar, yet conflicting mental models, syntaxes, programming tools, and modes of interaction. A task that overlaps the domains of two or more languages causes conceptual problems, because of conflicts between mental models, and requires

tricky, time-consuming, bug-prone interfaces. Since the user should only need to learn one set of programming tools, those tools cannot be specifically designed for any one language, but must compromise. For example, it is difficult to provide a editor that performs automatic syntax checking for all the programming languages.

3.3 How S fits in

To make efficient use of the machine (maximize the time spent in step 4 of the process life cycle) S needs to run in a process that persists indefinitely—to avoid wasted effort in starting and stopping multiple processes—and be adaptable enough to satisfy a data analyst's need for unpredictable improvization. To accomplish this, S, in effect, layers its own interactive programming environment on top of the Unix system. S is a tour-de-force of Unix programming and does an impressive job of providing a rich and flexible environment for data analysis. However, S has two major disadvantages: slow execution speed and the difficulty of adding new primitives.

The lack of speed results from (1) the lack of a compiler for the S language and (2) the fact that S macros and data are represented by files, so almost every operation requires I/O through the file system, rather than simply referring to procedures and data resident in memory. There is no compiler because writing a compiler is a major task and would have to be redone for every machine that S runs on. Macros and data are represented by files because it's difficult to do dynamic allocation of memory in languages like C and Fortran and writing a proper memory manager and garbage collector would be comparable in complexity to writing an S compiler. Using the Unix file system is an easy solution, if inefficient.

Some procedures cannot be written as S macros, most often because of a need for fast execution. In that case the user will have to add a new primitive to the S system. Introducing a new primitive is a major undertaking; the complexity of the process (see [6]) is a good example of the language interfacing problems mentioned in the previous section.

These problems in part result from the evolution of S over time, as its authors learned more about what they wanted the system to do, and could

be ameliorated by a redesigned, cleaned up version of S. (For example, a new system on Common Lisp [34] would be highly portable, be automatically be provided with an interpreter and compiler, the distinction between primitives and macros would go away, and dynamic memory allocation and garbage collection would be handled by the Lisp interpreter.)

We believe, however, that the problems could not be eliminated entirely, because they are symptoms of a more basic defect. S and similar, large sub-systems, like Macsyma, are each isolated from the rest of the operating system, which leads to a tendency to re-invent the wheel, that is, to make special purpose versions of programming tools that should be done in uniform, coherent way for the whole programming environment, not just for the part of the environment devoted to data analysis.

4 Integrated Programming Environments

Integrated programming environments have two features that distinguish them from a conventional operating systems and are significant for data analysis: all procedures and data reside in a *single, persistent address space* and all programming is done in a *single language.*

4.1 A single address space

Experimental programming involves incremental changes to existing data and procedures. In a conventional operating system, it is hard to make changes of small granularity. A program must be loaded into memory and all its data read in from the file system every time it is run—because files are the only objects that have an indefinite lifetime. Much of the work would be unnecessary if the procedures and data remained in memory between invocations of the program.

An integrated programming environment does exactly that; all procedures and data are located in a single address space that is shared by all processes and that persists indefinitely. For example, if a program creates and fills an array with numbers, then the array will still be in memory and

available to any future programs that are run. Since memory is finite, automatic garbage collection—to reclaim space occupied by procedures and data objects which are no longer needed—is an essential element of an integrated programming environment.

Function calls and data references go through a symbol table. Modifying a function definition is done by changing a single entry in the symbol table. Because no linking is necessary and because unchanged procedures do not have to be reloaded into memory, nearly all of the overhead in the most frequent programming operations—usually involving direct changes to one or two procedures—is eliminated. Simple, mechanical programming tasks are accomplished in 1/10 to 1/100 of the time they take in a conventional operating system, which is an overwhelming advantage for the type of exploratory, interactive programming that goes on in data analysis.

The single address space gives the programmer a great deal of—possibly dangerous—freedom. A programmer can usually redefine any system functions; a typical exercise is to redefine the basic arithmetic operations to handle vectors and matrices as well as numbers. This kind of freedom is clearly impractical in a multi-user operating system and perhaps not particularly useful for large team, production-style programming. However, a good integrated environment provides tools that help organize and manage the contents of the persistent address space and encourage the programmer to exercise an appropriate discipline.

4.2 A single language

One of the most striking differences between conventional operating systems and integrated environments is the fact that the same language is used used both as the interpreted command language (the shell) and the compiled programming language. This is possible, in part, because the persistent address space provides a context for interactively evaluating isolated expressions in the programming language. For example, executing the Unix command: `cc file.c` is equivalent to evaluating the expression `(compile a-function)` on a Lisp machine. The difference is that the `compile` is a Lisp function like any other and `a-function` is a Lisp data object. `compile`—and any other system command—can be called in a program in exactly the

same way it is used interactively. `a-function` can be manipulated by a user program in the same way that it is manipulated interactively. This means that the user has a real programming language (Lisp) for system operations, as opposed to the primitive command languages like the Unix shell.

The basic advantages of a monolingual environment are [20]:

- The user has only one model of the programming environment to understand.
- There are far fewer arbitrary, petty details to be remembered—like minor differences in syntax.
- Language interfacing problems are eliminated. (This is particularly important in a single address space system, where procedures and data are shared by all programs. Consider, for example, the problems that would arise in trying to refer to a Pascal record in a Fortran subroutine.)

The fact that we want to use a single language for all programming tasks implies several things about the kind of language that is used:

- Functions as data.

  Because all programming is done in the one language, all manipulation of programs—definition, compilation, loading, and execution—must be controlled within the language itself. Thus it is necessary to be able to treat function objects as data, to pass them to other functions as arguments, and to return them as computed values. This is true in Lisp and Smalltalk, but only true to a limited degree in languages like C or Pascal.

- Data abstraction and run-time typing.

  Most statistical computing in done in a Fortran style—creating and modifying procedures that operate on a limited set of data structures (numbers and arrays). In more modern languages, the programmer in

encouraged to devote an equal or greater amount of time to the definition and modification of new types of data structures. A language that encourages experimentation with data structure definitions in a persistent address space should support *data abstraction* and *run-time typing.*

Data abstraction means that it is possible to define generic procedures that do the "right thing" when called with varying types of arguments. For example, the plus function should behave properly when given integers, floating point numbers, complex numbers, matrices, and even a code for missing data or a representation of infinity. Data abstraction is a fundamental part of Smalltalk and well supported in modern dialects of Lisp, but very difficult in Pascal, Fortran, or C.

A language that supports data abstraction can either require types of all variables to be defined at compile-time (as in Mesa) or it can allow procedures to determine the types of their arguments at run-time. For experimental programming, run-time typing is essential, otherwise modifying the definition of a data structure would require recompiling all the procedures that might refer to it.

For example, one might like to try out several alternatives for a representation of large sparse matrices. It should be possible to use a library of linear algebra routines without having to modify or recompile them to deal with each variation. (What would be needed is generic array element look-up functions.)

The need for efficient run-time typing is a major reason why Lisp and Smalltalk machines benefit from non-standard architectures. A common technique is to encode some type information in *tag bits* in each machine word. For example, a machine with 8 tag bits and 32 data bits (to allow for IEEE standard format floating point) would have a word length of be 40 bits instead of the 32 bits words found in microprocessors like the Motorola 68k. The instruction set would also have to be specialized to permit quick extraction and analysis of the tag and data fields.

- Integrated dialects.

> Because there are legitimately distinct types of problems, the language should allow the definition of dialects that provide abstractions appropriate for a particular domain [11,20]. However, if done properly, a *monolingual* environment will have a coherent mental model, a consistent syntax, a properly partitioned set of programming tools, and predictable modes of interaction for all dialects.
>
> A principal reason for the popularity of Lisp is its ability to support different styles of programming within a consistent framework. Many examples of high level abstraction in Lisp are given by Abelson and Sussman [1]. Examples of specialized languages built on top of Lisp are Flavors, [9] an object-oriented extension of Common Lisp, and Loops [8,35], an extension of Interlisp-D that supports object-oriented, data-oriented, rule-based, and procedure-oriented styles. Runtime typing, data abstraction, and treating functions as data are essential elements in building these extensions to Lisp.

The monolingual environment has an obvious defect—existing software written in other languages is not available. For example, statisticians benefit from the extensive numerical analysis libraries (e.g. Linpack) written in Fortran.

One solution is to choose an environment that is not strictly monolingual, such as that provided on the Symbolics Lisp machines. Such systems provide compilers for more traditional languages like Fortran, C, and Pascal and an interface that allows procedures in these languages to be called from Lisp programs. The intent is that essentially all new programs will be written in the primary language the environment is designed for (e.g. Lisp). Some tools may be provided for programming in the auxiliary languages, their purpose is mostly to help the programmer avoid re-inventing the wheel. However, introducing additional languages inevitably leads to some of the difficulties inherent in multilingual environments.

In pure monolingual environments (perhaps Smalltalk), the only alternative is to translate or re-code whatever is needed. We feel strongly that this alternative is not as bad as it seems to most statisticians at first glance.

Most statisticians underestimate the difficulty of transporting and using programs written in standard Fortran, even ones as well-documented

and carefully written as Linpack. At the same time, they overestimate both the difficulty of re-writing and the amount of code that needs to be re-written. The classical numerical analysis libraries were developed in a punchcard/batch-processing programming environment that is orders of magnitude slower at mechanical programming tasks than the integrated environments. Since the numerical algorithms are well understood (which was perhaps not quite true at the time the libraries were first written), the programming needed to re-implement them is the type that is accelerated most. It is, for example, possible to write and debug a singular value decomposition in a few hours [29].

Still, re-coding standard numerical analysis routines would be at best uninteresting and a comprehensive recoding would require a substantial investment of time. A more serious drawback is that recoded libraries will not be as thoroughly tested and reliable.

5 Conclusion

The purpose of this paper has been to argue that, for *research* in data analysis, especially research in data analysis systems, a workstation with an integrated programming environment is a better choice than one with a conventional operating system. The principal reason is that the low level structure of an integrated environment is more hospitable to the improvizational, experimental programming that is an essential part of good data analysis. Another important point is that the languages used (Lisp and Smalltalk) allow the data analyst to choose abstractions that more closely fit the problems to be solved. For our own research, we have chosen Lisp machines with support for object-oriented programming (e.g. Flavors, on the Symbolics and Loops on the Xerox machines).

We should make it clear that we are not suggesting that any of the existing integrated programming environments are suitable for analyzing real data. No system like S is available for these environments yet. The best choice for doing data analysis at present is to get a workstation with a conventional operating system and S (or possibly ISP [12,13]). Recent versions of S and ISP explore some of the possibilities of graphics workstations, for

example, by including high-interaction techniques like rotating scatterplots and scatterplot matrices.

It is difficult to predict how long it will be before a system with functionality as comprehensive as S is available on a Lisp machine. Work on S began in the mid-70's. It started to appear outside Bell Labs about 5 years later and became fairly widespread in academic institutions after about 10 years of development. We might, on the one hand, expect development of a similar system on a Lisp machine to take less time, because elementary programming tasks get done much more quickly.

On the other hand, integrated environments represent a significant departure from the past and it will take time to explore and understand their novel possibilities. An early example of this exploration is the DINDE system of Oldford and Peters [28]. For the similar reasons, the Lisp environments themselves are the subject of active research and therefore somewhat unstable; examples are the on-going research in object-oriented programming and window systems. This inhibits the development of a stable statistical system. However, it has a positive aspect as well. The structure of integrated environments is not frozen; if statisticians start research now, they can influence the standards as they develop.

5.1 A poor man's integrated environment

A reasonable, intermediate research goal would be to produce an improved version of S, with the following properties:

- One language that can be used as the shell, macro, or compiled implementation language, which will imply fast execution, simpler internal structure, and easier addition of new functions.
- Automatic management and garbage collection of a persistent virtual memory which will allow faster access to statistical data and functions.
- Portable over many types of workstations and operating systems.

- A representation of statistical data in the data structures of the language.
- As comprehensive functionality as S.
- Device independent static graphics.

We propose that any new system of this type should be implemented in Common Lisp [34]. Common Lisp systems (including compiler, interpreter, and some programming tools like an editor and debugger) are available for a large and growing number of workstations and operating systems. the first three requirements mentioned above come for free with Common Lisp. The remaining three would be added by the implementor. Most Common Lisp implementations provide mechanisms for calling routines in other languages, so statistical functionality can be provided in the same way it is done in S, using standard Fortran libraries. Providing device independent static graphics is a more substantial undertaking, though some standards exist.

References

[1] ABELSON, H., SUSSMAN, G., and SUSSMAN, J. (1985)
Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass.

[2] BAWDEN, A., GREENBLATT, R., HOLLOWAY, J., KNIGHT, T., MOON, D., and WEINREB, D. (1979)
The LISP Machine in Artificial Intelligence: An MIT Perspective. vol.II. WINSTON, P.H. and BROWN, R.H., eds.

[3] BARSTOW, D.R., SHROBE, H.E., and SANDEWALL, E., eds. (1984)
Interactive Programming Environments. McGraw-Hill, New York.

[4] BECKER, R.A. and CHAMBERS, J.M. (1984a)
Design of the S System for Data Analysis CACM 27 (5): 486-495.

[5] BECKER, R.A. and CHAMBERS, J.M. (1984b)
S: An Interactive Environment for Data Analysis and Graphics. Wadsworth, Belmont, Ca.

[6] BECKER, R.A. and CHAMBERS, J.M. (1985)
Extending the S system. Wadsworth, Belmont, Ca.

[7] BELL SYSTEM (1978)
a special issue on Unix, The Bell System Technical Journal. 57.

[8] BOBROW, D.G. and STEFIK, M. (1983)
The LOOPS Manual. Xerox PARC, Palo Alto, Ca.

[9] CANNON, H.I. (1982)
Flavors—A non-hierarchical approach to object oriented programming, manuscript from Symbolics, Inc., 5 Cambridge Center, Cambridge, Mass. 02142.

[10] DEITEL H.M., (1983)
An Introduction to Operating Systems. Addison-Wesley, Reading, Mass.

[11] DEUTSCH, L.P., and TAFT, E.A, eds., (1980)
Requirements for an Experimental Programming Environment, Xerox Parc Tech Report CSL-80-10.

[12] DONOHO, D., HUBER, P.J., and THOMA, H. (1981)
The Use of Kinematic Displays to Represent High Dimensional Data, Computer Science and Statistics: Proc. of the 13th Symposium on the Interface. Edited by Eddy, W.F., New York, Springer-Verlag, 1981

[13] **Donoho, D., Huber, P.J., Ramos, E.** and **Thoma, H.** (1982)
Kinematic Display of Multivariate Data,
Proc. of the Third Annual Conference and Exposition of the National Computer Graphics Association, Inc., Volume I.

[14] DORADO (1981)
The Dorado: A High Performance Personal Computer, Three Papers, Xerox PARC Report CSL-81-1.

[15] FOLEY, J.D. and VAN DAM, A. (1982)
Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading, Mass.

[16] GABRIEL, R.P. (1985)
Performance and Evaluation of Lisp Systems. MIT Press, Cambridge, Mass.

[17] GOLDBERG, A. (1983)
The Influence of an Object-Oriented Language on the Programming Environment reprinted in BARSTOW, D.R., SHROBE, H.E., and SANDEWALL, E., eds. (1984)
Interactive Programming Environments.

[18] GOLDBERG, A. (1984)
Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, Mass.

[19] GOLDBERG, A. and ROBSON, D. (1983b)
Smalltalk-80. The Language and Its Implementation. Addison-Wesley, Reading, Mass.

[20] HEERING, J., and KLINT, P. (1985)
Towards Monolingual Programming Environments, ACM Trans. on Programming Languages and Systems. 7. pp. 183-213.

[21] KERNIGHAN, B.W. and MASHEY, J.R. (1981)
The Unix Programming Environment, Computer 14 (4): 25-34.

[22] KERNIGHAN, B.W. and PLAUGER, P.J. (1976)
Software Tools. Addison-Wesley, Reading, Mass.

[23] KERNIGHAN, B.W. and RITCHIE, D.M. (1978)
The C Programming Language. Prentice-Hall, Englewood Cliffs, N.J.

[24] KRASNER, G., ed. (1983)
Smalltalk-80. Bits of History. Words of Advice. Addison-Wesley, Reading, Mass.

[25] MCDONALD, J.A. and PEDERSEN, J.O. (1985)
Computing Environments for Data Analysis, Part I. Introduction, SIAM J. Scientific and Statistical Computing 6 (4): 1004-1012.

[26] MCDONALD, J.A. and PEDERSEN, J.O. (1985)
Computing Environments for Data Analysis, Part II. Hardware, SIAM J. Scientific and Statistical Computing 6 (4): 1013-1021.

[27] MOSTELLER, F. and TUKEY, J.W. (1977)
Data Analysis and Regression. Addison-Wesley, Reading, Mass.

[28] OLDFORD, R.W. and PETERS, S.C. (1986)
DINDE: Towards more statistically sophisticated software, unpublished manuscript.

[29] PETERS, S.C. (1986)
personal communication.

[30] SANDEWALL, E. (1978)
Programming in an Interactive Environment: the "LISP" Experience, ACM Computing Surveys 10.

[31] SHEIL, B.A. (1983)
Power Tools for Programmers reprinted in BARSTOW, D.R., SHROBE, H.E., and SANDEWALL, E., eds. (1984)
Interactive Programming Environments.

[32] SHEIL, B.A. (1983)
Environments for Exploratory Programming, in *Papers on Interlisp-D,* SHEIL, B.A. and MASINTER, L.M., eds., Xerox PARC Tech Report CIS-5.

[33] SHEIL, B.A. and MASINTER, L.M., eds. (1983)
Papers on Interlisp-D, Xerox PARC Tech Report CIS-5.

[34] STEELE, G.L. (1984)
Common Lisp The Language. Digital Press.

[35] STEFIK, M., BOBROW, D.G., MITTAL, S. and CONWAY, L. (1983)
Knowledge Programming in LOOPS: Report on an experimental Course, The AI Magazine 3: 3-13.

[36] SYMBOLICS (1983)
3600 Technical Summary. Symbolics, Inc. 5 Cambridge Center, Cambridge, Mass. 02142.

[37] TEITLEMAN W. and MASINTER, L. (1981)
The Interlisp Programming Environment, reprinted in BARSTOW, D.R., SHROBE, H.E., and SANDEWALL, E., eds. (1984)
Interactive Programming Environments.

[38] THACKER, C. et al. (1979)
Alto: a personal computer, Xerox PARC Technical Report CSL-79-11.

[39] TUKEY, J.W. (1977)
Exploratory Data Analysis. Addison-Wesley, Reading, Mass.

[40] XEROX (1983)
Interlisp Reference Manual. October 1983. Xerox Corporation.

A Appendix: an example

The remainder of this paper will illustrate the comparison between conventional operating systems and integrated programming environments by working a concrete example.

By improvizational data analysis, we don't mean to suggest that the user will be constantly developing profound, new approaches to data analysis—we have in mind modest, but unpredictable innovations, something like the following.

Suppose we have been drawing histograms using some pre-defined procedures and data structures, without bothering to understand all their details. We decide we'd like to modify the existing procedures (and data structures, if necessary) to be able to draw logograms as well.

If a histogram is a bar chart where the heights of the bars are proportional to the counts in the corresponding bins, then the logogram is a bar chart where the heights are proportional to the log of the counts. Although the logogram is probably a bad choice for a data analyst (a rootogram is usually prefered), we use it as an example because it lets us introduce a trivial error–what happens if we take the log of zero?

A.1 A sequence of actions

To go from a histogram to logogram, we need to:

- Find out about the current state of the system.

  In other words, we need to know what data structures and procedures are used to tabulate the histogram and draw it.

  Let's suppose that drawing a histogram uses three types of data structures and two procedures: (a) a (statistical) database, (b) a structure holding the counts for each bin of the histogram, and (c) a structure representing the plot in which the histogram is drawn; (a) a procedure which queries the database and tabulates the counts in the histogram structure and (b) a procedure that draws a plot based on a historam structure.

- Figure out what changes to make to get desired effect.

  In this case, the most obvious choices would be to either (a) modify the tabulation procedure so that the resulting histogram data structure would hold logs of the counts rather than the counts or (b) modify the drawing procedure so that it draws bars whose heights are proportional to the logs of the counts rather than the counts themselves. Alternative (a) might, in some programming languages, require modifying the histogram data structure so that its cells can hold floating point numbers rather than counts.

- Make the changes easily and correctly.

  We need to find and edit the source code that defines the procedures and data structures mentioned above, without introducing errors into any existing code and without wasting time on trivial syntax errors in our own additions.

- Translate into executable form.

  Finally we need to compile, link, and load our modifications into the existing system so that we can use the logogram on the data set that inspired us to program it in the first place.

A.2 Logograms in Unix

For ease of presentation, we present the example as though we did data analysis under Unix by executing shell commands from a predefined library of statistical functions.

A more reasonable scenario would be to assume that we did interactive analysis using a statistical language like S. For this particular example, it would be much easier to write an S macro than to incrementally modify a C program. However, S is not a complete answer; if high speed were needed, say, for interactive graphics, the logogram would have to be added as a primitive function. We recommend this to the reader as a good illustration of the language boundary problems that arise in multilingual environments.

Basically, the reason that it is easier to do data analysis under S is that—for problems that can be solved with macros—S offers an approximation to an integrated environment.

Suppose our machine has a builtin library of statistical functions that can be called interactively from the Unix shell. To use the statistics commands we need to include the directory `/usr/stat/bin` in our search path, which we do by typing:

```
set path = (/usr/stat/bin $path).
```

The statistics library sends plots to a device defined by a shell environment variable `STAT_G_DEV`, which we can set by:

```
setenv STAT_G_DEV = /dev/gwin0,
```

which might mean to send the plots to graphics window zero.

To compute and draw a histogram, we type:

```
hist < file.data.
```

The execution time will, of course, depend on the size of the batch. For small data sets (100's of points), it should take at most 10 seconds. Note, however, that the time is spent reading the data from the file system must be redone every time we compute a new histogram.

So far, we are using the shell language, `csh`. Because `csh` was not originally intended as a programming language, it developed petty inconsistencies, like the difference in syntax between changing the search path and setting the shell variable.

Suppose that, after looking at the first plot, we decide we would like to try a "logogram" instead. We will need to find out more about how the stat library works.

We list `/usr/stat` and find that it has a subdirectory called `src`. In `/usr/stat/src`, we find `Makefile`, which is a specification, written in the

```
STAT    = /usr/stat
SOURCES = $(STAT)/src
HEADER  = $(STAT)/def/statlib.h
STATLIB = $(STAT)/lib/statlib.a
BIN     = $(STAT)/bin
...
$(BIN)/hist: $(SOURCES)/hist.o $(STATLIB)
cc -o $(BIN)/hist $(SOURCES)/hist.o $(STATLIB) -lm
$(SOURCES)/hist.o: $(HEADER)
cc -c -O $(SOURCES)/hist.c
...
$(STATLIB): $(BIN)/statlib.o $(HEADER)
ar r $(STATLIB) $(BIN)/statlib.o
ranlib $(STATLIB)
```

Figure 1: The make file for the stat library.

make language, for compile and link dependencies among a group of files. It is shown in figure 1.

If we decipher this, we see that the `hist` command is defined in `/usr/stat/src/hist.c` which is shown in figure 2.

To make sense of `hist.c`, we mentally switch to a third language, namely C. It's not hard to guess that `Batch`, `Histogram`, and `DisplayList` are data structures used by the statistics library. The procedure `GetBatch` collects the data, `TabulateHistogram` does the counting work, and `PlotHistogram` and `DrawPlot` produce the picture.

There are two alternative ways to proceed. One is to write a new procedure, `TabulateLogogram`, to replace `TabulateHistogram`. The other is to replace `PlotHistogram` by a `PlotLogogram`. Tabulation seems simpler than drawing plots, so we decide to look at `TabulateHistogram` (figure 3). To find its source code, we type `grep TabulateHistogram *.c` in various subdirectories of `/usr/stat` and finally find it in `/usr/stat/lib/summar.c` (see figure 3).

```
#include <stdio.h>
#include <math.h>
#include "/usr/stat/defs/statlib.h"

main()
    {
    Batch       the_data;
    Histogram   the_histogram;
    Displaylist the_plot;

    GetBatch(&the_data, stdin);
    the_histogram.nbins = 0;
    TabulateHistogram(&the_data, &the_histogram);
    MakePlotHistogram(&the_plot, &the_histogram);
    DrawPlot(&the_plot);
    }
```

Figure 2: hist.c

```
TabulateHistogram(batch, hist)
Batch     *batch;
Histogram *hist;
    {
    int i;

    InitializeHistogram(batch, hist);
    for(i=0; i<batch->size; i++)
       {
       hist->cells[GetBin(hist,batch->data[i])] += 1;
       }
    }
```

Figure 3: The definition of TabulateHistogram

`hist->cells` is an array that holds the counts. What we would like to do is just insert a final loop that replaces `hist->cells[i]` by `log(hist->cells[i])`. Unfortunately, `hist->cells` is an array of integers. This means that we cannot put floating point numbers in it because the compiler won't let us and, more importantly, because all the routines that use a `Histogram` data structure will expect the `cells` to hold integers. Modifying the definition of `Histogram` would possibly require editing all the procedures that use it and certainly require recompiling all of them. The stat library is large (1,000's of lines) and we are not familiar with its internal details, so making this change correctly would take days, if not weeks. This is an example of the difficulties that result from lack of data abstraction and run-time typing.

Instead, we decide to approximate `log(hist->cells[i])` by the closest integer to `100*log(hist->cells[i])`. This means that the labeling of the plot will be incorrect. It's also the wrong way to start out if we intend to build further on the `logogram`, for example, if we wanted to add error bars around the heights of the bars. However, the shape of the plot will be roughly correct.

We create a file `loghist.c` by copying `/usr/stat/src/hist.c` (using the `csh` language). We enter a screen editor (another language and mental model) and add the procedure `TabulateLogogram`, by copying `TabulateHistogram` from `/usr/stat/lib/summar.c` and then inserting a final loop:

```
for(i=0; i < hist->nbins; i++)
   {
   hist->cells[i] = 100 * log( (float) hist->cells[i]);
   }
```

When we are done, we exit the editor environment and go back to the shell environment to compile new version by typing:

```
cc -o loghist -O loghist.c /usr/stat/src/statlib.a -lm
```

Then we run it by typing `loghist file.data`.

The first time, the result is:

```
Floating Point exception (core dumped)
```

In this example, it's not too hard to figure out what's wrong, even without any debugging at all.

The quality of debuggers varies greatly across versions of Unix. In a system with a poor debugger, the only reliable way to find out what is going wrong is to liberally insert print statements to trace the execution.

Many Unix systems provide a source level debugger called dbx, which is yet another language and mental model. To use it we need to re-compile with a special -g flag:

```
cc -g -o loghist loghist.c /usr/stat/src/statlib.a -lm
```

We enter dbx and type run; dbx answers:

```
floating point exception at line 55
       hist->cells[i] = 100 * log( (float) hist->cells[i]);
```

We then type `print hist->cells[i]` which returns 0. The error is fixed by replacing `log(hist->cells[i])` by `log(hist->cells[i] + 1)`, after which we edit, compile, and try again.

Note that we have used 5 overlapping languages and models of the environment: `csh`, `make`, C, an editor, and `dbx`.

A.3 Logograms in Interlisp-D

Interlisp-D Interlisp-D is a Lisp-based integrated environment developed at Xerox PARC and runs on the Xerox 1100 series of workstations. It is the product of over ten years of development and features a residential Lisp environment, a large collection of program analysis and debugging tools, a simple yet powerful window system, and a substantial library of applications software, including several graphical editors.

Interlisp-D is designed for use on a single-user graphics workstation. A typical configuration includes a Xerox custom built, microprogrammable

bit-sliced processor with between 1.5 and 3.5 Megabytes of main memory, a 1024 x 780 high-resolution monochrome bitmap display, a three button mouse pointing device, a 40 Megabyte local hard disk, a floppy-disk drive, and a 10 Megabit ethernet connection to the outside world. A floating point processor option is available in the form of a Weitek 1032/1033 high performance chip set which is rated at between 1.1 and 5 Megaflops. From Interlisp-D, in some situations, it possible to get floating point performance superior to the DEC VAX 11/780. The Xerox 1100 series of workstations, all of which run identical versions of Interlisp-D at differing speeds, is comprised of five machines ranging in price from $10,000 to $80,000, although a development configuration usually costs between $20,000 and $40,000.

Interlisp-D is a monolingual environment based on a dialect of Lisp called Interlisp. All aspects of the environment are coded in this language, from high-level commands to low-level system internals, so that there is no fundamental distinction between applications code and what might be called the "operating system". The programmer's model of the Interlisp-D environment is a large (32 Megabyte) persistent virtual memory space populated by data objects and programs. All actions occur within this virtual memory space, which is sometimes called a world-load. At startup the world-load consists of a default collection of roughly 4 Megabytes worth of data objects and programs which might be called the Interlisp-D kernel. The world-load may be augmented by creating new objects or loading new objects into the environment from secondary storage and, on occasion, the world-load is backed up by writing some portion of it to secondary storage, but the state of the machine is described solely by the contents of the world-load, which evolves over time. The model is very similar to that of APL environments, for those familiar with that language.

The Interlisp-D kernel provides standard programming tools, such as a Lisp interpreter and compiler, a Lisp code editor, peripheral device drivers, memory management and network access software, along with tools customized for the Lisp workstation environment, such as a window system, a network-wide file system, browsing and organizational facilities for the virtual memory space, and static and dynamic Lisp program analysis tools.

Interlisp-D depends heavily on the window system to organize information on the high-resolution display. The display is divided into rectangular

regions called windows, each of which represents a separate activity or context. Since the display is bitmapped, text and graphics may be freely intermixed within each window. Windows are objects in the environment which may overlap, may be created, destroyed, reshaped, moved, buried, redisplayed, either manually or under program control, and may be customized for particular activities. A window manager controls interaction, determining the target for graphical or keystroke input and keeping track of which windows are displayed, where the cursor is, etc. In combination with menus, which are specialized windows dedicated to command selection, the window system allows the user to interact graphically with the environment and is a crucial aspect of the user interface to Interlisp-D. At startup, a few default windows are displayed including a top-level Lisp listener and a message display window. As the session progresses window are created and discarded as needed. In the midst of a session the display may appear as in figure 4.

The window labeled Interlisp-D Executive evaluates Lisp expression typed in at the keyboard, and is often called a Lisp-listener. At the top of the screen is an icon, which represents a window that is still active, but compressed and moved out of the way. The tiled window labeled Lafite is the electronic mail browser. Off to the right hand side are a history window, which maintains a record of expressions typed into the Lisp-listener, and a manager window which describes the currently loaded collections of procedures and data structures. In the foreground is a plot window displaying a histogram. The plot window overlaps two editor windows.

The example To draw a histogram the user types the Lisp expression (HISTPLOT DATA) where DATA is a variable bound to a data object which already exists in the environment (in this case a list of numbers). The result of this expression a plot data object which is then available for further computation.

As can be seen from the screen image in figure 4, the plotted data has a large peak at zero. To produce a logogram the user needs to examine the function definition of HISTPLOT. This is accomplished by typing the Lisp expression (DF HISTPLOT) which causes an editor window to appear as in

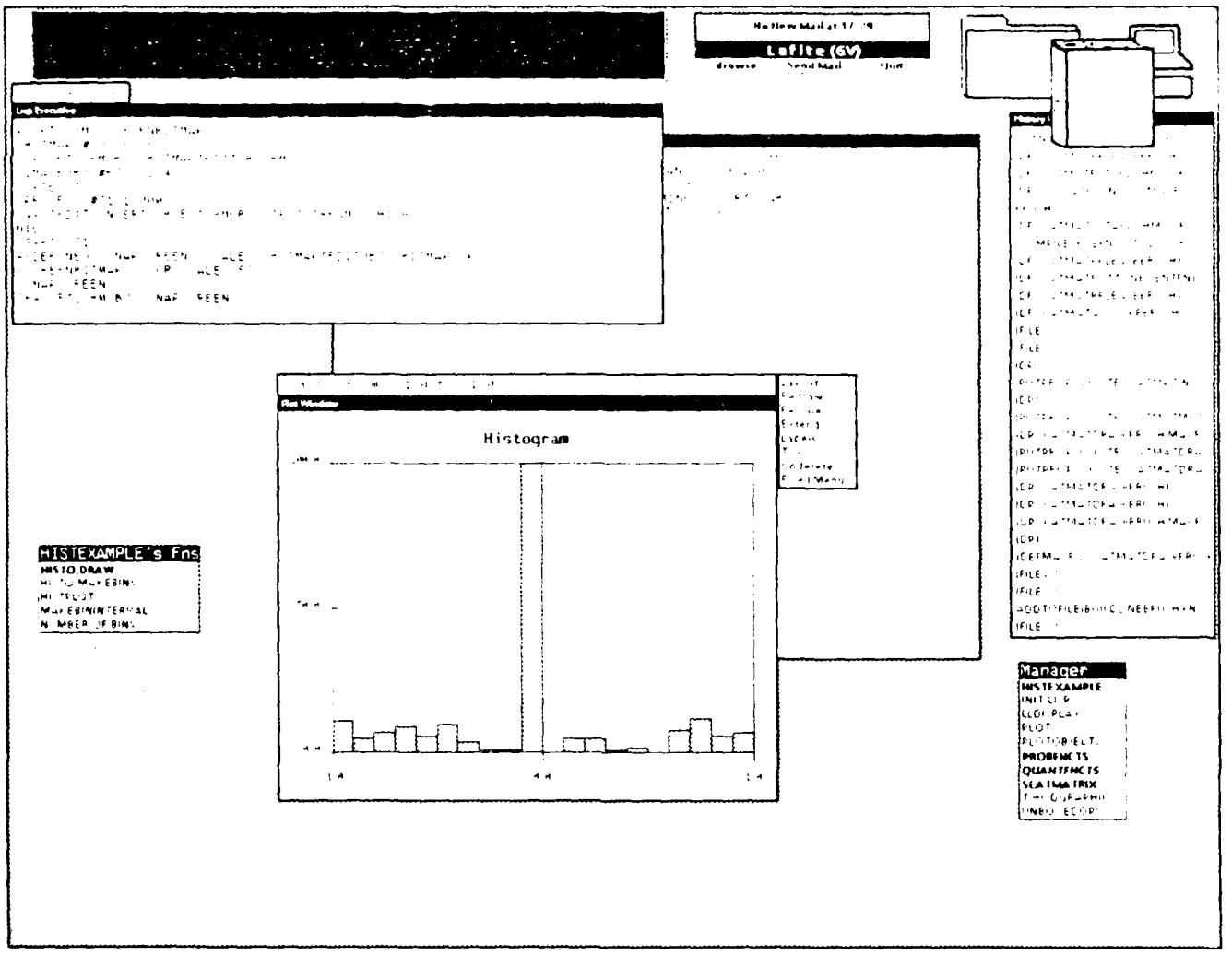


Figure 4: The workstation screen as it might appear while working the example.

figure 5. The Interlisp-D editor, is a menu and mouse driven Lisp structure editor. That is, all operations are specified on the menu attached to the right of the editor window, and the editor has full knowledge of Lisp syntax; for example, it is impossible to produce unbalanced parentheses with this editor.

The function HISTPLOT calls several subsidiary functions, the most interesting of which are HISTO.MAKEBINS and HISTO.DRAW. To edit one of those functions, the user could simply select (with the mouse) the function name and button the Edit item in the editor menu, to produce another editor window. However, instead of browsing the function definitions in this manner, the user may prefer to generate a global view of the interconnections between functions in HISTPLOT. Interlisp-D has a tool called called Masterscope which may be used to generate such a view.

Masterscope analyses the functions currently loaded (a fairly straight forward task in Lisp) and maintains a data base of information. Masterscope may then be queried for objects in the environment that satisfy certain relations. For example, the user may ask to edit all functions which call the function HIST.MAKEBINS. Masterscope may also be asked to display graphically the tree of function calls from a specified root function. In this example, the user might ask Masterscope to show the call structure from HISTPLOT. The result of this operation is a grapher window as shown in figure 6. The nodes in the represented tree structure are active. For example, If the cursor is pointed at one of the nodes and the left button is depressed then an editor window pops up with the appropriate function definition. The window is also scrollable to the right. That is, the portion of the graph which is not currently shown and extends off to the right may be viewed by moving the cursor to the bottom of window and buttoning right.

It is clear from the grapher window that HISTO.DRAW's job is simply to display the histogram, calling many graphics functions in the process, while HISTO.MAKEBINS actually does the binning. The user might now edit the definition of HISTO.MAKEBINS by buttoning the HISTO.MAKEBINS node in the grapher window to produce an editor window as in figure 7.

The function definition is moderately complex; however the sorted and binned numbers are passed back as a property of the histogram. The rel-

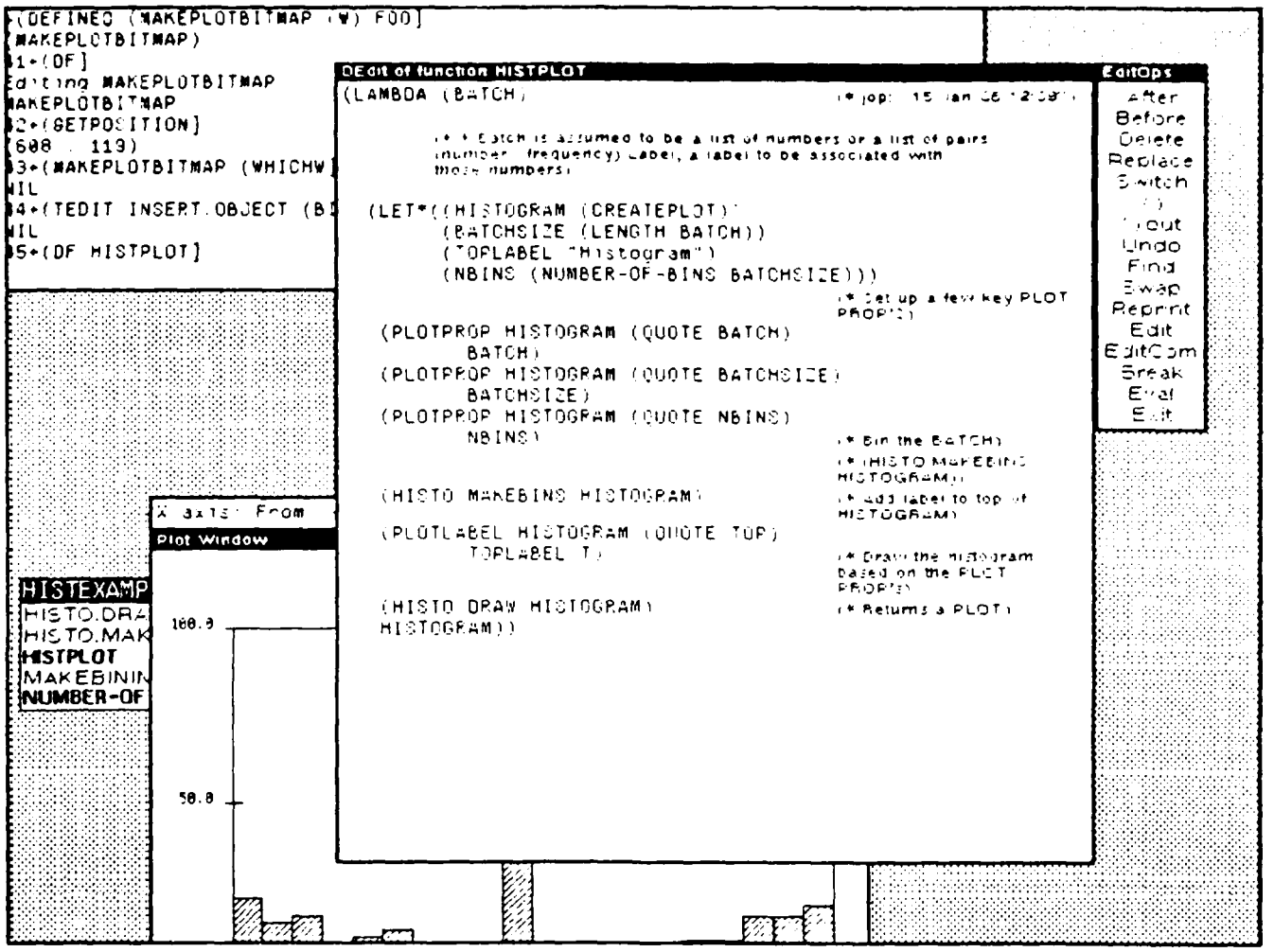


Figure 5: An expanded view of the screen including an editor window.

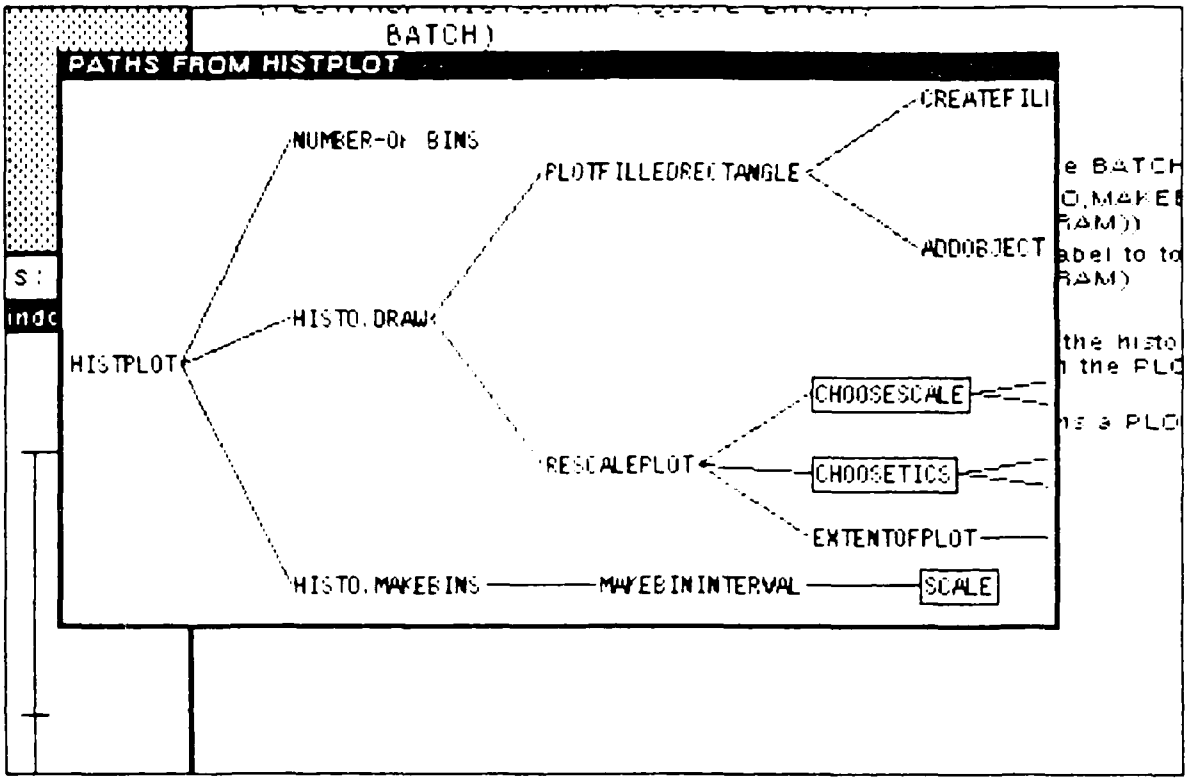


Figure 6: grapher window showing call struction from the function HIST-PLOT.

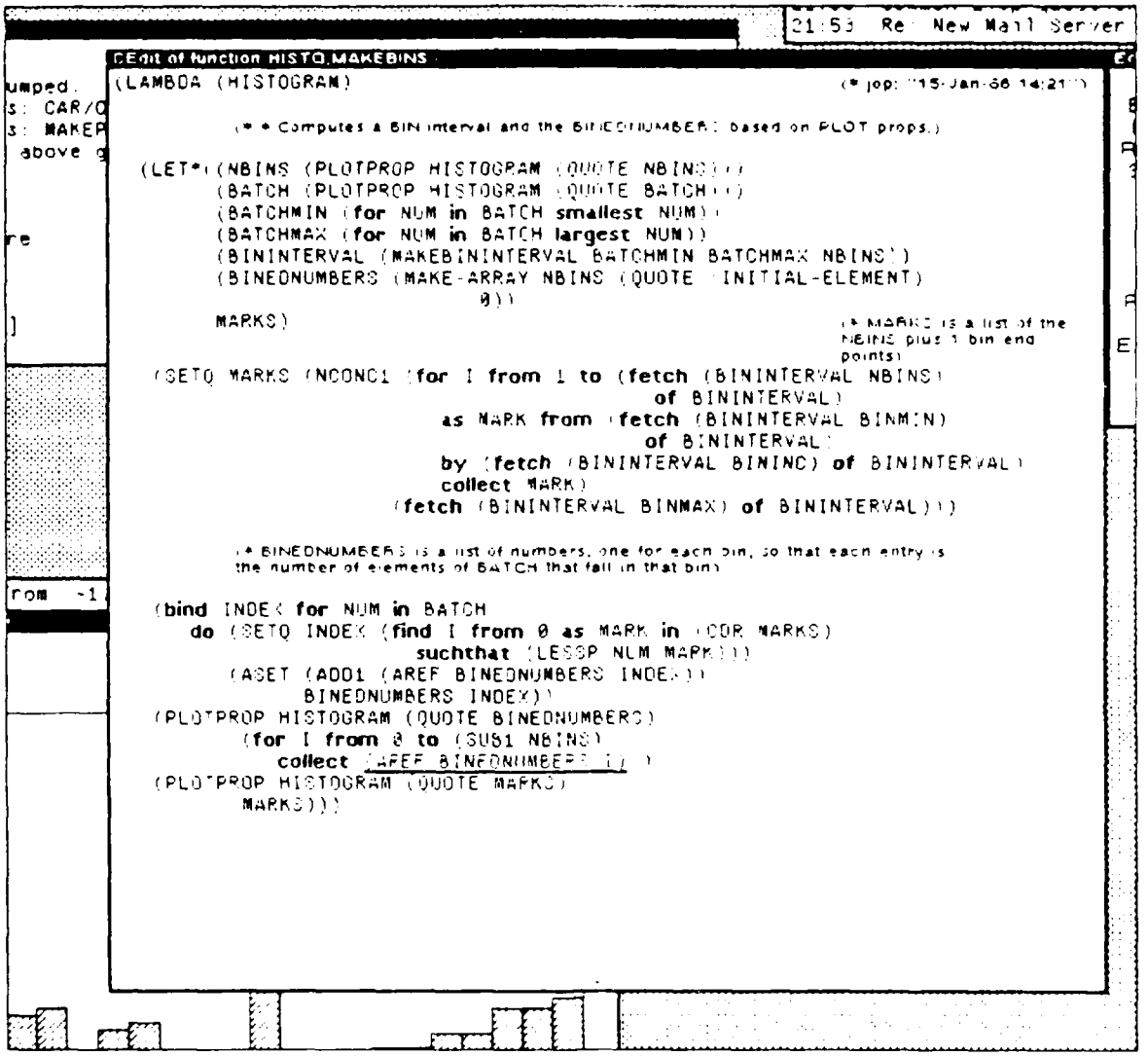


Figure 7: Editor window showing the function definition of HISTO.MAKEBINS.

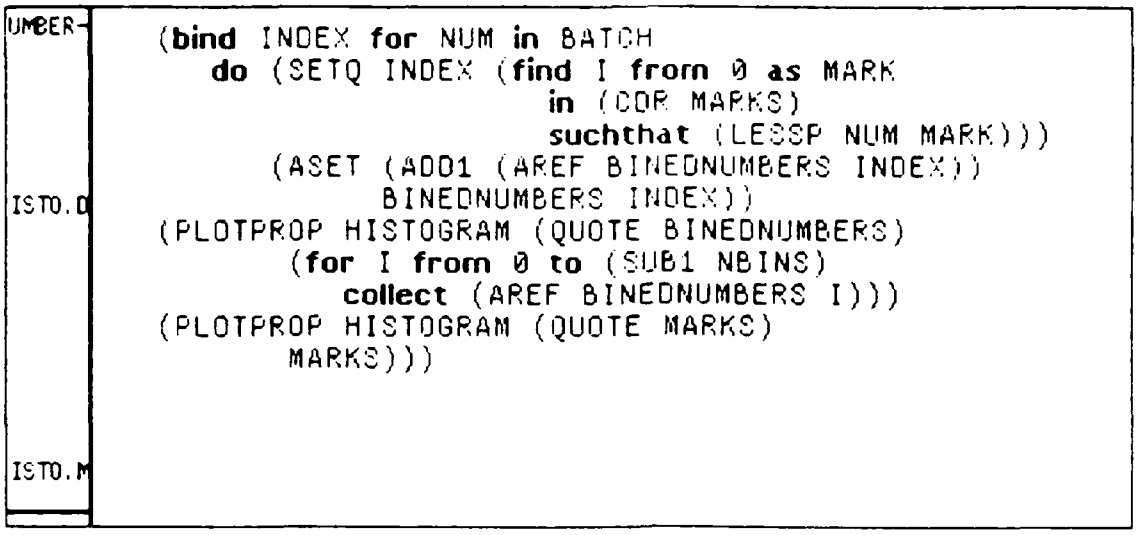


Figure 8: Closeup of a critical part of HISTO.MAKEBINS.

evant lines are shown in figure 8. The histogram has a property called BINEDNUMBERS which describes of the frequency in each bin. The property called MARKS describes the end points for each bin.

It is sufficient to apply log's to the frequency in each bin to produce a logogram. This change is effected by the insertion of an additional function call, as shown in figure 9

Note that it is not necessary to convert the frequencies to floating point numbers before applying LOG. This is an instance generic arithmetic which is possible because LOG may query the type of its argument at run time. Also, the result of this expression will now be a list a floating point numbers, rather than integers, but due to generic arithmetic we can be fairly confident that this will not disrupt the rest of the computation.

To instantiate the change the user simply exits the editor. The system now overrides the previous definition of HISTO.MAKEBINS and the user is immediately able to try out the new definition by typing (HISTPLOT DATA) in the lisp-listener window.

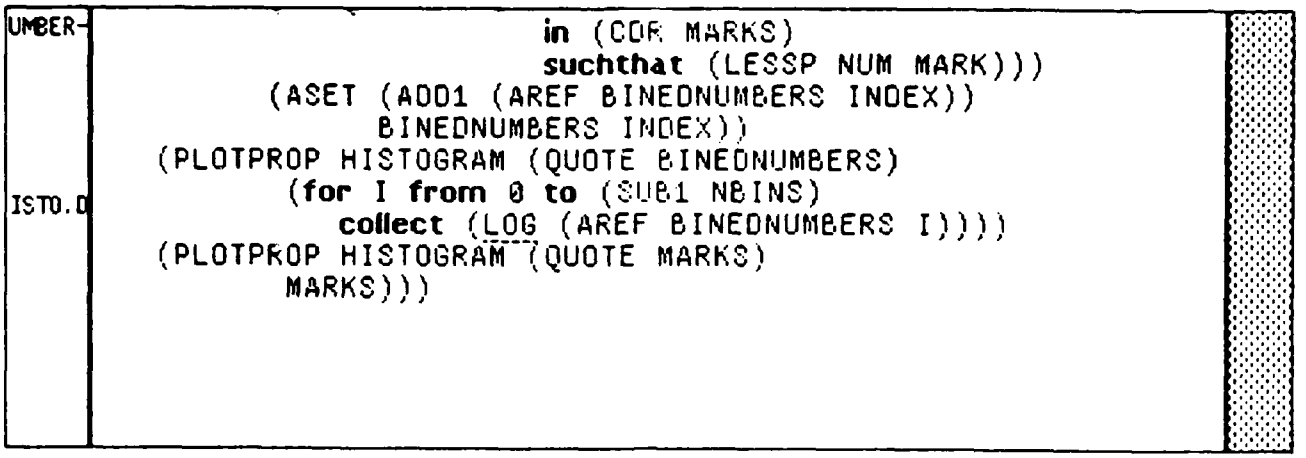


Figure 9: Closeup of editor window after change to HISTO.MAKEBINS.

The function HISTPLOT evaluates until an error occurs, in this case an attempt to take the log of zero. The computation is suspended at the point of the error and a break window pops up detailing the type of error and in which function it occurred, as show in figure 10. The break window is also a lisp-listener, so Lisp expression may be evaluated in the context of the break. For example, to examine the value of the variable HISTOGRAM, it is sufficient to type its name in the break window. To the right of a break window is a menu describing the execution stack at the time of the error. For example, the attempted log of zero occurred below HISTO.MAKEBINS. The HISTO.MAKEBINS entry has been selected and a representation of that stack frame is displayed above the break window. The local variables used in that stack frame are displayed with their names and values. For convenience, the values of variables may be inspected or set from the stack frame window. The function definition associated with the stack frame may be edited by selecting its name in the stack frame window, as has been done in the illustration.

The error clearly occurred in the function HISTO.MAKEBINS, and the user might now edit that function from the break window, by selecting its name from the execution stack menu. Since some bins will have zero

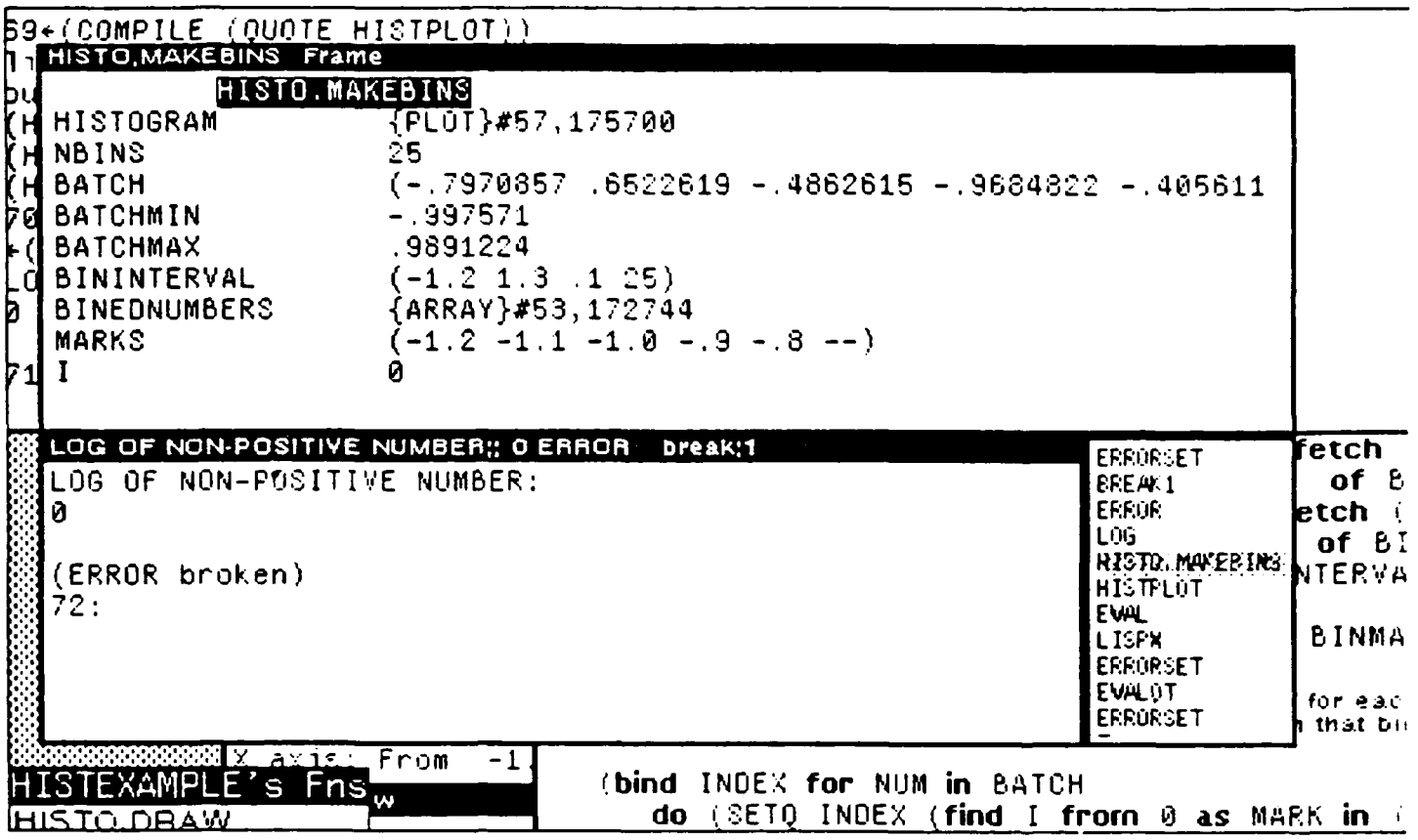


Figure 10: Closeup break window generated from evaluation of logogram function.

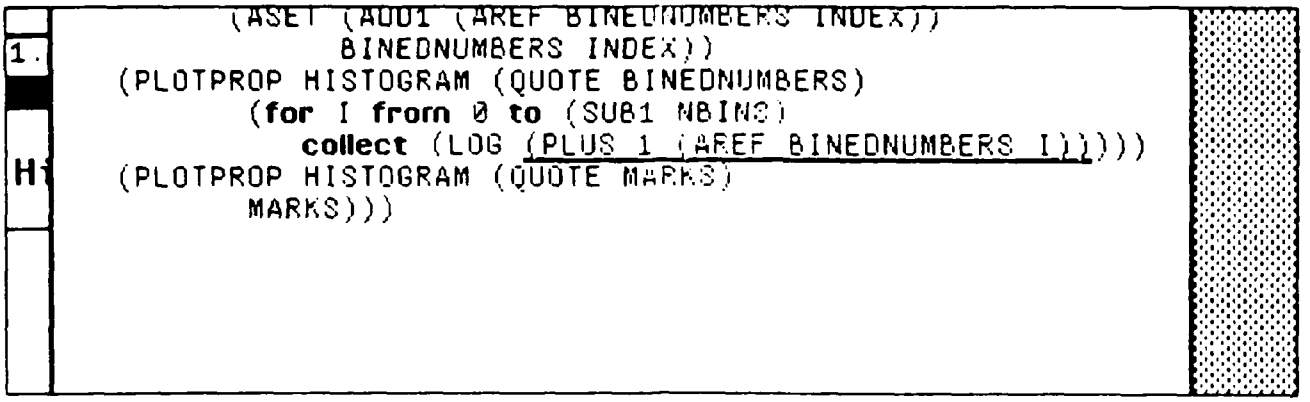


Figure 11: Closeup of change to definition of HISTO.MAKEBINS.

counts, the user decides to add one to the frequencies for each bin. This may be effected by inserting an expression before evaluating LOG, as show in figure 11.

Since the context of the error is captured in the break window, the computation may now be restarted from the error. Alternatively, the user may abort the computation and start again from the top. In either case the computation proceeds without error to produce the logogram as shown in figure 12

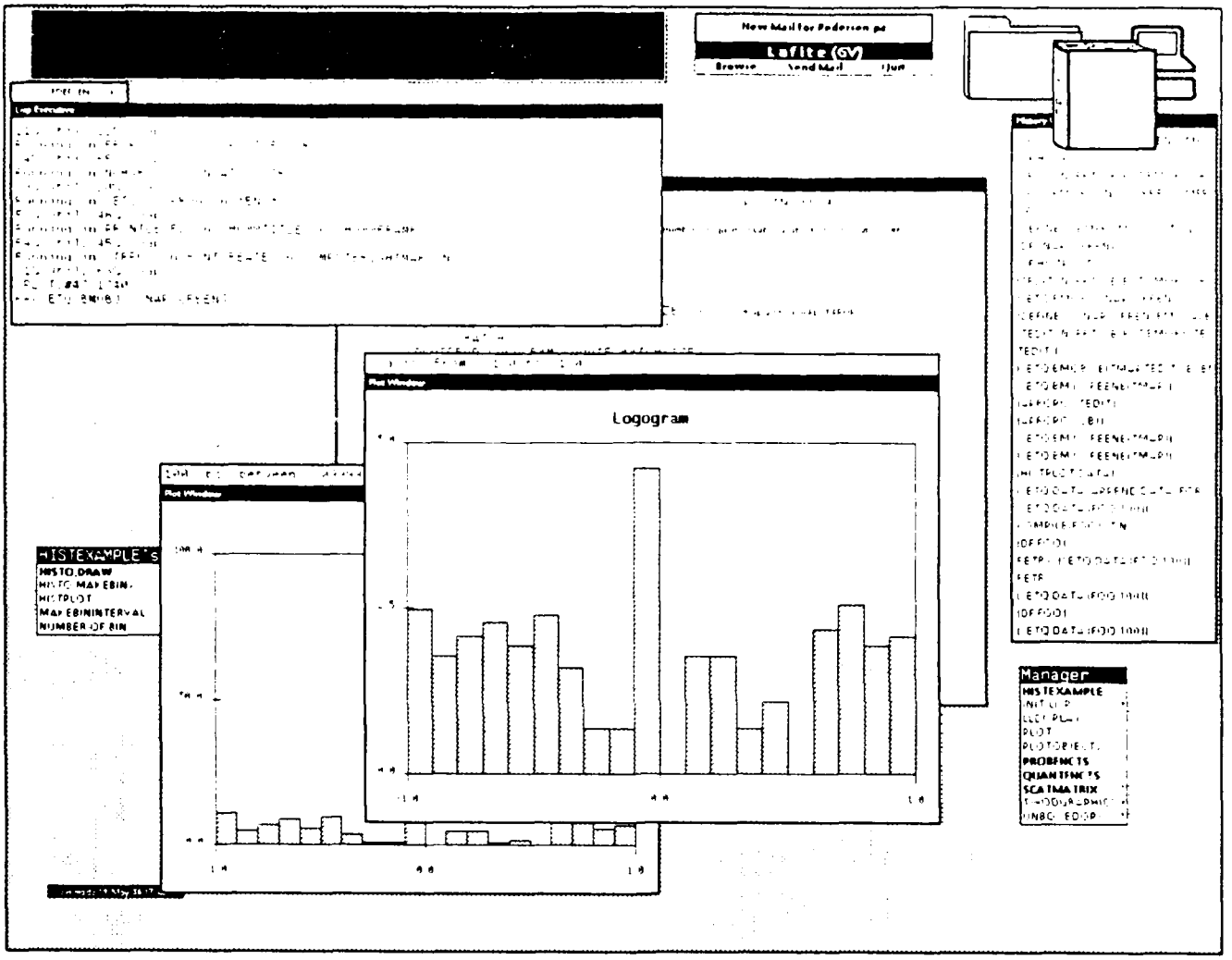


Figure 12: Image of the screen after completion of example. The logogram window is in the foreground partially covering the previous histogram.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 24	2. GOVT ACCESSION NO.: AD-A168 848	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Computing Environments for Data Analysis: Part 3: Programming Environments	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): John A McDonald and Jan Pedersen	8. CONTRACT OR GRANT NUMBER(s): N00014-83-G-0121
9. PERFORMING ORGANIZATION NAME AND ADDRESS: U.S. Office of Naval Research, Department of the Navy, Arlington, VA 22217	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: —	12. REPORT DATE: May 1986
	13. NUMBER OF PAGES: 43
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Navy position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data analysis, workstations, programming environments

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the third in a series of papers on aspects of modern computing environments that are relevant to statistical data analysis. In this paper, we discuss programming environments. In particular, we argue that 'integrated programming environments' (for example, Lisp and Smalltalk environments) are more appropriate as a base for data analysis than conventional operating systems (for example Unix).

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Laboratory for Computational Statistics

Technical Reports

1. Friedman, J.H., Smart User's Guide, October 1984.
2. Hastie, T.J. and Tibshirani, R.J., Generalized Additive Models, September 1984.
3. Tibshirani, R.J., Bootstrap Confidence Intervals, October 1984.
4. Tibshirani, R.J., Local Likelihood Estimation, September 1984.
5. Friedman, J.H., A Variable Span Smoother, November 1984.
6. Owen, A., Nonparamteric Likelihood Ratio Intervals, December 1985.
7. McDonald, J.A. and Owen, A.B., Smoothing with Split Linear Fits, October 1984.
8. Marhoul, J.C. and Owen, A.B., Consistency of Smoothing with Running Linear Fits, November 1984.
9. McDonald, J.A. and Pedersen, J., Computing Environments for Data Analysis: Part I: Introduction, April 1984.
10. McDonald, J.A. and Pedersen, J., Computing Environments for Data Analysis: Part 2: Hardware, April 1984.
11. Hastie, T., Principal Curves and Surfaces, November 1984.
12. Friedman, J.H., Classification and Multiple Regression through Projection Pursuit, January 1985.
13. Marhoul, J.C., A Model for Large Sparse Contingency Tables, December 1984.
14. Efron, B., Better Bootstrap Confidence Intervals, November 1984.
15. Segal, M., Recursive Partitioning Using Ranks, August 1985.
16. Johns, M. V., Fully Nonparametric Empirical Bayes Estimation via Projection Pursuit, August 1985.
17. Altman, N.S., Expert Systems and Statistical Expertise, Part I: Statistical Expert Systems, May 1985.
18. Friedman, J.H., Exploratory Projection Pursuit, November 1985.
19. Efron, B. and Tibshirani, R., The Bootstrap Method for Assessing Statistical Accuracy, October 1985.
20. Hjort, N.L., Bayesian Nonparametric Bootstrap Confidence Intervals, November 1985.
21. Hjort, N.L., Bootstrapping Cox's Regression Model, November 1985.
22. Hjort, N.L., On Frequency Polygons and Average Shifted Histograms in Higher Dimensions, February 1986.
23. Scott, D.L. and Terrell, G.R., Biased and Unbiased Cross-Validation in Density Estimation, April 1986.
24. McDonald, J.A. and Pedersen, J., Computing Environments for Data Analysis: Part 3: Programming Environments, May 1986.

END

DTIC

7-86